Međuispit iz OSZUR-a, 24. 4. 2012.

_____

ime i prezime

1. a) (3 boda) Nadopuniti slijedeći kod potrebnim ključnim riječima (`include`, `static`, ...) tako da se kod može prevesti naredbom: `gcc main.c device1.c device2.c -o d2d`.

"device.h"

```
struct device_t {
        int (*init) ();
        int (*recv) ( void *data, size_t size );
        int (*send) ( void *data, size_t size );
}
```

"main.c"

```
#include "device.h"
extern device_t device1, device2;

#define M       80
int main () {
        char buffer[M];
        size_t size;
        device1.init();
        device2.init();
        while(size = device1.recv ( buffer, M ))
                device2.send ( buffer, size );
        return 0;
}
```

"device1.c"

```
#include "device.h"
static  int init () { ... }
static  int recv ( void *data, size_t size )
        { ... }
static  int send ( void *data, size_t size )
        { ... }

struct device_t device1 = (struct device_t)
{ .init = init, .recv = recv, .send = send };
```

"device2.c"

```
#include "device.h"
static  int init () { ... }
static  int recv ( void *data, size_t size )
        { ... }
static  int send ( void *data, size_t size )
        { ... }

struct device_t device2 = (struct device_t)
{ .init = init, .recv = recv, .send = send };
```

b) (2 boda) Napisati *Makefile* za prevođenje gornjih datoteka.

c) (2 boda) Navesti izlazne odjeljke koji će se pojaviti prevođenjem gornjih datoteka te navedite sadržaje tih odjeljaka (koji elementi gornjih datoteka će biti u njima).

Makefile:

```
d2d: main.o device1.o device2.o
        gcc main.o device1.o device2.o -o d2d
main.o: main.c device.h
        gcc -c main.c
device1.o: device1.c device.h
        gcc -c device1.c
device2.o: device2.c device.h
        gcc -c device2.c
```

.text => sve instrukcije
.data => device1, device2
.bss => buffer i size (na stogu)
eventualno: .rodata => 80 (ali to se najčešće ugradi u instrukciju

2. (3) Napisati makroe (sa `#define IME`) naziva `INC1(N)`, `INC2(N,X)` te `INC3(N,X)` tako da:
   * `INC1(N)` vraća vrijednost za jednu veću od `N`,
   * `INC2(N,X)` vraća vrijednost za jednu veću od `N` ako je `N<X-1` te 0 inače, te
   * `INC3(N,X)` koji povećava varijablu `N` za jedan ako je `N<X-1`, odnosno postavlja ju u 0 inače.

   Makroe napisati tako da budu uporabljivi u svim primjenama (kontekstu) koje imaju smisla (poslani parmetri N i X mogu biti i složeniji izrazi; sam makro može biti dio složenijih izraza, primjerice `INC1` može se koristiti u `INC2` a `INC2` u `INC3`). Po potrebi koristiti "uvjetno" dodjeljivanje:

   `( uvjet ? vrijednost_za_DA : vrijednost_za_NE ).`

```
#define INC1(N)      ( (N) + 1 )
#define INC2(N,X)    ( (N) < (X)-1 ? INC1(N) : 0 )
#define INC3(N,X)    do { N = INC2(N,X); } while (0)
```

3. (2) Neki zamišljeni procesor ima 4 registara opće namjene `R0-R3` te programsko brojilo `PC`, registar stanja `RS` i kazaljku stoga `SP`. Za rad sa stogom ima instrukcije `PUSH registar` i `POP registar` koje stavljaju zadani registar na stog i obnavljaju vrijednost registra sa stoga. Pri prihvatu prekida procesor sam stavlja na stog `PC` i `RS`. Ukoliko sve prekida treba obraditi funkcijom obradi_prekid (`CALL obradi_prekid`), te ukoliko se iz prekida vraćamo instrukcijom `IRET` (koja obnavlja `RS` i `PC` sa stoga i omogućuje prekide) napisati niz instrukcija koje slijede labelu `prihvati_prekid` a koje se izvode po prihvatu prekida (procesor nastavlja obradu prekida tim instrukcijama nakon što je sam na stog pohranio `PC` i `RS`).

```
prihvati_prekid:

        PUSH R0
        PUSH R1
        PUSH R2
        PUSH R3

        CALL  obradi_prekid

        POP R3
        POP R2
        POP R1
        POP R0

        IRET
```

4. (8 bodova) Ostvariti podsustav za upravljanje vremenom koji omogućuje postavljanje jednog alarma (jedina funkcionalnost). Neka sučelje koje treba ostvariti bude:

   `postavi_alarm ( vrijeme_do_aktiviranja, funkcija ).`

   Nakon isteka zadanog vremena (`vrijeme_do_aktiviranja`, u mikrosekundama, računanog od trenutka postavljanja alarma – poziva `postavi_alarm`) treba pozvati funkciju `funkcija`. Na raspolaganju stoji brojilo koje odbrojava u taktu jedne mikrosekunde, a čija se vrijednost (u mikrosekundama) postavlja sa `postavi_brojilo ( int broj )` (sa `broj=0` se brojanje isključuje) i čita sa `pročitaj_brojilo ( int *broj )` (na adresu `broj` se upisuje trenutna vrijednost brojila). Po dostizanju vrijednosti nula, brojilo izaziva prekid `PREKID_BROJILA` koji se može povezati funkcijom za obradu prekida pozivom `registriraj_prekid ( ID_PREKID, funkcija_obrade )`. Neka se podsustav, tj. sve funkcije koje ga sačinjavaju, od `postavi_alarm`, `obrada_prekida_sata` te `inicijaliziraj()`, kao i sve potrebne varijable nalaze u datoteci `alarm.c`. Napisati sadržaj te datoteke. Radi jednostavnosti vrijeme izražavati u mikrosekundama i pretpostaviti da neće doći do prekoračenja opsega brojeva tipa `int` pri njegovu korištenju za tu svrhu te da je brojilo dovoljno veliko da prihvati sve intervale.

alarm.c

```
static void (*fun) ();

static void obrada_prekida_sata ()
{
      void (*f2) ();
      postavi_brojilo (0); //nije neophodno
      f2 = fun;   //zbog mogućeg ponovnog postavljanja alarma u obradi;
      fun = NULL; //ali nije neophodno za bodove
      if ( f2 != NULL )
            f2 ();
}

void inicijaliziraj ()
{
      fun = NULL;
      postavi_brojilo (0); //nije neophodno
      registriraj_prekid ( PREKID_BROJILA, obrada_prekida_sata );
}

void postavi_alarm ( vrijeme_do_aktiviranja, funkcija )
{
      fun = funkcija;
      if (  vrijeme_do_aktiviranja > 0 )
            postavi_brojilo (  vrijeme_do_aktiviranja );
      else
            obrada_prekida_sata (); //aktiviraj odmah
}
```

1. [2 boda] Na zadanome kodu označiti koji dio (varijable, kod) će se staviti u koji odjeljak pri prevođenu (koristiti samo `.text`, `.data` i `.bss` odjeljke).

```
#include "zaglavlje.h"
struct nesto[10];                          .bss
int a = 3;                                 .data
static int b = 5;                          .data
int main () {
    int x, y;                              .bss (ili drugdje, na stogu)
    x = a * 5; y = x * a;                  .text
    b += funkcija1 ( x, y, nesto );        .text
    a += funkcija2 ( nesto );              .text
    return a + b;                          .text
}
```

2. [2 boda] Zadan je makro:

```
#define LOG(LEVEL,FORMAT,...) \
fprintf ( log, #LEVEL FORMAT "\n", ##__VA_ARGS__ )
```

Ako se on pozove sa: `LOG ( A, "%d", a );` u što će se pretvoriti makro u početnoj fazi prevođenja (engl. *preprocessing*)?

Rj.: `fprintf ( log, "A%d\n", a );`

3. [2 boda] Napisati makro `KVJ(A,B,C,X1R,X1I,X2R,X2I)` za izračunavanje rješenja kvadratne jednadžbe: `A*x^2 + B*x + C = 0`. Pretpostaviti da su parametri realni brojevi te da `A` nije nula. Ulazni parametri: `A`, `B` i `C` mogu biti i složeni izrazi pa i povratne vrijednosti funkcija (npr. `KVJ(5+f1(),3,get(a),x1r,x1i,b,c)`).

Rj.:

```
#define KVJ(A,B,C,X1R,X1I,X2R,X2I)          \
do {                                        \
    double a = (A), b = (B), c = (C);       \
    double korjen = b*b - 4*a*c;            \
    if ( korjen >= 0 ) {                    \
        korjen = sqrt(korjen);              \
        X1R = (-b - korjen) / 2 / a;        \
        X2R = (-b + korjen) / 2 / a;        \
        X1I = X2I = 0;                      \
    }                                       \
    else {                                  \
        korjen = sqrt(-korjen);             \
        X1R = X2R = -b / 2 / a;             \
        X1I = -korjen / 2 / a;              \
        X2I = -X1I;                         \
    }                                       \
}                                           \
while(0)
```

4. [2 boda] Neki sustav se sastoji od datoteka: `a.h`, `a.c`, `b.h`, `b.c` te `main.c`. Odgovarajuće `.c` datoteke koriste odgovarajuća zaglavlja tj. `.h` datoteke, dok `main.c` koristi oba zaglavlja. Pri prevođenju datoteke `a.c` treba koristiti zastavicu `-DZ1`, za `b.c` zastavicu `-DZ2` te za `main.c` zastavice `-DZ3 -DZ4`. Pri povezivanju (engl. *linking*) treba postaviti zastavicu `-lnesto`. Napisati `Makefile` kojim će se moći izgraditi zadani sustav u program naziva `test1`.

Rj.: Makefile

```
test1: a.o b.o main.o
	gcc -o test1 a.o b.o main.o -lnesto

a.o: a.c a.h
	gcc -c a.c -DZ1

b.o: b.c b.h
	gcc -c a.c -DZ2

main.o: main.c a.h b.h
	gcc -c main.c -DZ3 -DZ4
```

5. [2 boda] Čemu služe ključne riječi: **`extern`**, **`static`**, **`inline`** i **`volatile`**? Opisati njihovo korištenje na primjerima.

6. [2 boda] Zadan je algoritam dinamičkog upravljanja spremnikom kod kojeg su slobodni blokovi u LIFO listi, tj. kod kojeg se pri oslobađanju bloka i nakon njegova eventualna spajanja sa susjednim on u listu slobodnih blokova doda na početak liste. Navesti dobra i loša svojstva tog algoritma.

7. [2 boda] Čemu služi nadzorni alarm (engl. *watchdog timer*)?

8. [2 boda] Neki sustav treba pripremiti za učitavanje u ROM na adresi `0x10000`. Podaci (odjeljci `.data` i `.bss` će se pri pokretanju kopirati na adresu `0x100000` te ih (podatke) treba pripremiti za tu adresu (ali učitati u ROM). Napisati skriptu za povezivača (engl. *linker*) koja će omogućiti navedeno. U skriptu dodati potrebne varijable.

Rj.: skripta.ld

```
ROM = 0x10000;
RAM = 0x100000;
SECTIONS {
	.kod ROM:
	{
		* (.text .rodata)
	}
	podaci_pocetak = .;
	.podaci RAM : AT ( ROM + SIZEOF(.text) )
	{
		* ( .data .bss )
	}
	podaci_kraj = podaci_pocetak + SIZEOF(.podaci);
}
```

9. [3 boda] Neki ugrađeni sustav ima tri naprave. Prve dvije N1 i N2 treba poslužiti iz obrade prekida funkcijama `n1()` i `n2()` (te funkcije postoje), dok se sa N3 upravlja programski – u petlji glavnog programa, pozivom `n3()`. Naprava N1 neće ponovno generirati zahtjev za prekid dok prethodni zahtjev te naprave nije obrađen do kraja. Isto vrijedi i za napravu N2. N3 ne generira zahtjeve za prekid. Naprava N1 jest najvažnija i njene zahtjeve treba najmanje odgađati (tj. ne odgađati). Funkcije `n1()`, `n2()` i `n3()` mogu trajati proizvoljno dugo (prema potrebi u pojedinome trenutku). Sustav posjeduje prekidni podsustav sa sučeljem:

```
registriraj_prekid ( id, funkcija );
zabrani_prekidanje ();
dozvoli_prekidanje ();
```

Pokazati ostvarenje funkcija `x_n1()` i `x_n2()` te `glavni_program()` u koje će se postaviti dodatne potrebne operacije prije poziva `n1()`, `n2()` i `n3()` (prema potrebama).

Rj.:

```
glavni_program () {
    registriraj_prekid ( N1, x_n1() );
    registriraj_prekid ( N2, x_n2() );
    dozvoli_prekidanje ();
    ponavljaj {
        n3();
    }
}
x_n1() {
    n1(); //obrada prekida sa zabranjenim prekidanjem;
}
x_n2() {
    dozvoli_prekidanje ();
    n2(); //obrada prekida s dozvoljenim prekidanjem;
    zabrani_prekidanje ();
}
```

Zadaci 10. i 11. jesu opširniji, ali oni donose "dodatne" bodove – MI nosi 20% a ovim se zadacima može ostvariti 25 bodova (5 "bonus" bodova).

10. [3 boda] Ostvariti podsustav za upravljanje vremenom sa sučeljem:

```
int dohvati_sat ( int *sek, int *usek );
int postavi_sat ( int sek, int usek );
int postavi_alarm ( int sek, int usek, void (*funkcija)() );
```

Za ostvarenje na raspolaganju stoji brojilo dohvatljivo na adresi `0x8000` koje odbrojava frekvencijom od 1 Mhz. Najveća vrijednost koja stane u brojilo je 10^9. Kada brojilo dođe do nule izazove prekid i stane. U obradi tog prekida pozove se funkcija `prekid_brojila()` (koju treba napraviti, pored gornjih). Vrijeme u sekundama i mikrosekundama je relativno u odnosu na neki početni trenutak (nebitno koji).

Rj.: (jedno od)

```
#define MAXCNT 1000000000
#define TICKSPERSEC     1000000
int zadnje_ucitano = MAXCNT;
int *brojilo = (int *) 0x8000;
int sat_sec = 0, sat_usec = 0;
int alarm_sec = 0, alarm_usec = 0;
void (*alarm) () = NULL;

int postavi_sat ( int sek, int usek )
{
    //provjere preskočene ( sek >= 0 && usek >= 0 && usek < 1000000 )
    sat_sec = sek;
    sat_usec = usek;
    alarm = NULL; //poništava se alarm (napomenuto na ispitu)
    zadnje_ucitano = MAXCNT;
    *brojilo = zadnje_ucitano;
    return 0;
}

int dohvati_sat ( int *sek, int *usek )
{
    //provjere preskočene ( sek != NULL && usek != NULL )
    *sek = sat_sec + ( zadnje_ucitano - *brojilo ) / TICKSPERSEC;
    *usek = sat_usec + ( zadnje_ucitano - *brojilo ) % TICKSPERSEC;
    if ( *usek >= TICKSPERSEC ) {
        *usek = *usec - TICKSPERSEC;
        *sek = *sek + 1;
    }
    return 0;
}
//nastavak na idućoj strani
```

```
int postavi_alarm ( int sek, int usek, void (*funkcija)() )
{
    // provjere preskočene:
    // ( sek >= 0 && usek >= 0 && usek < 1000000 && funkcija != NULL )

    //relativan alarm: za {sek,usek} ga aktiviraj
    alarm_sec = sek;
    alarm_usec = usek;

    alarm = funkcija;

    prekid_brojila ();
}

void prekid_brojila ()
{
    //ažuriraj sat
    sat_sec += ( zadnje_ucitano - *brojilo ) / TICKSPERSEC;
    usec += ( zadnje_ucitano - *brojilo ) % TICKSPERSEC;
    if ( usec >= TICKSPERSEC ) {
        usec -= TICKSPERSEC;
        sec++;
    }

    zadnje_ucitano = MAXCNT;
    *brojilo = zadnje_ucitano;

    if ( alarm != NULL ) {
        if ( alarm_sec + alarm_usec == 0 ) {
            void (*tmp)() = alarm;
            alarm = NULL;
            tmp ();
        }
        else {
            if ( alarm_sec < MAXCNT / TICKSPERSEC )
                zadnje_ucitano = alarm_usec + alarm_sec * TICKSPERSEC;
            //else zadnje_ucitano = MAXCNT; -- već prije postavljeno

            //koliko još ostane za idući puta?
            alarm_sec -= zadnje_ucitano / TICKSPERSEC;
            alarm_usec -= zadnje_ucitano % TICKSPERSEC;
            if ( alarm_usec < 0 ) {
                alarm_sec--;
                alarm_usec += TICKSPERSEC;
                if ( alarm_sec < 0 ) {
                    //greškica u nepreciznosti; idući prekid je alarm
                    alarm_sec = 0;
                    alarm_usec = 0;
                }
            }
        }
    }
}
```

11. [3 boda] U nekom sustavu sučelje za rad s napravama jest:

```
struct naprava_t {
    int (*init) ( struct naprava_t *n );
    int (*send) ( struct naprava_t *n, void *data, size_t size );
    int (*recv) ( struct naprava_t *n, void *data, size_t size );
    void *param;
};
```

Napisati upravljački program za napravu X korištenjem gornjeg sučelja. Pretpostaviti da je naprava dostupna na adresama S (za slanje), R (za čitanje) i C (za statusni registar). Čitanjem podatka na adresi C dobiva se status naprave. Ukoliko je prvi bit pročitanog broja postavljen onda se s adrese R može pročitati idući podatak (ima ga). Ukoliko je drugi bit postavljen može se napravi poslati novi podatak (ona će ga moći prihvatiti). Radi ubrzanja rada za ulaz i izlaz dodati međuspremnike kapaciteta 4096 B (rezervirati ih s `malloc()`) i koristiti ih za pohranjivanje novih podataka iz naprave, odnosno, za privremenu pohranu kada se podaci ne mogu proslijediti prema napravi. Operacije `send` i `recv` trebaju koristiti te međuspremnike (u prethodno opisanim situacijama).

Rj.: (jedno od)

```
#define MS          4096
#define int8        unsigned char

struct ms {
    int8 bi[MS], bo[MS];
    int bi_f, bi_l, bi_sz, bo_f, bo_l, bo_sz;
};

static void x_interrupt_handler ( struct naprava_t *n );

static int x_init ( struct naprava_t *n )
{
    n->param = malloc (sizeof (struct ms) );
    memset ( n->param, 0, sizeof (struct ms) );
    registriraj_prekid ( X, x_interrupt_handler, n );
}

void x_interrupt_handler ( struct naprava_t *n )
{
    struct ms *ms = n->param;
    int8 *s = (int8 *) S, *r = (int8 *) R, *c = (int8 *) C;

    while ( ms->bi_sz < MS && ( (*c) & 1 ) ) {
        ms->bi[ms->bi_l] = *r;
        ms->bi_sz++;
        ms->bi_l = ( ms->bi_l + 1 ) % MS;
    }
    while ( ms->bo_sz > 0 && ( (*c) & 2 ) ) {
        *s = ms->bo[ms->bo_f];
        ms->bo_sz--;
        ms->bo_f = ( ms->bi_f + 1 ) % MS;
    }
}
//nastavak na idućoj strani
```

```
int x_send ( struct naprava_t *n, void *data, size_t size )
{
    struct ms *ms = n->param;
    int8 *d = data, sz = size;
    int8 *s = (int8 *) S, *c = (int8 *) C;

    //prvo probaj poslat izravno na napravu
    while ( ms->bo_sz == 0 && ( (*c) & 2 ) && sz > 0 ) {
        *s = *d;
        d++;
        sz--;
    }
    //ostatak u ms
    for ( ; sz > 0 && ms->bo_sz < MS; ) {
        ms->bo[ms->bo_l] = *d;
        d++;
        sz--;
        ms->bo_sz++;
        ms->bo_l = ( ms->bo_l + 1 ) % MS;
    }

    if ( sz > 0 )
        return size - sz; //toliko je ukupno poslano i stavljeno u ms
}

int x_recv ( struct naprava_t *n, void *data, size_t size )
{
    struct ms *ms = n->param;
    int8 *d = data, sz = size;
    int8 *r = (int8 *) R, *c = (int8 *) C;

    //prvo čitaj iz ms
    for ( ; ms->bi_sz > 0 && sz > 0; ) {
        *d = ms->bi[ms->bo_f];
        d++;
        sz--;
        ms->bi_sz--;
        ms->bi_f = ( ms->bi_f + 1 ) % MS;
    }
    //sada probaj čitat izravno s naprave
    while ( ( (*c) & 1 ) && sz > 0 ) {
        *d = *r;
        d++;
        sz--;
    }

    if ( sz > 0 )
        return size - sz; //toliko je pročitano
}

/* sučelje */
struct naprava_t x = { .init = x_init, .send = x_send, .recv = x_recv };
```

Rješenje zadataka (jedno od ...)

1. [1 bod] Gdje se sve nalaze kopije jedne datoteke koja je u sustavu koji koristi *git*? Pretpostaviti da se radi o jednom projektu – jednom git repozitoriju na poslužitelju te jednog korisnika koji ga koristi (dohvatio ga je sa `git clone ...`).

   Rj.
   a) u repozitoriju na poslužitelju
   b) u lokalnom repozitoriju (.git direktoriju)
   c) u lokalnoj kopiji ("radna inačica")

2. [1 bod] Radi provjere ispravnog rada u kod se ugrađuju dodatne provjere. Neke od njih se izvode samo u ispitnom pokretanju ('DEBUG' načinu), a neke uvijek. Pokazati na primjeru potrebu korištenja oba načina provjera.

   Rj.
   Ispitno pokretanje izvodi se pri razvoju sustava, dok još možda postoje neke greške u kodu (iako vjerojatno postoje i kasnije). Stoga se u tim pokretanjima koriste dodatne direktive pri prevođenju (DEBUG) te se uključuju dodatne provjere, pogotovo na početku funkcija dodatno se provjeravaju poslani parametri. Primjeri takvih ispitivanja su:

```
int neka_funkcija ( tip1 p1, tip2 p2, ... ) {
    ASSERT ( "provjera p1" ); //ili assert
    ...
}
```

   U fazi pravog rada sustava i dalje se mogu provjeravati neki parametri, pogotovo oni koji dolaze iz programa. Takve provjere moraju biti ostvarene običnim kodom (ne makroima ASSERT i sl.). Dodatne provjere tijekom pravog rada uglavnom se svode na probleme nedostatka sredstava i grešaka u radu (koje najčešće nisu rezultat krivog programa već ulaza i sl.). Primjerice, svaki bi zahtjev za stvaranje nekog objekta (npr. dretve) ili zahtjev za dijelom spremnika (malloc) trebalo provjeriti.

```
int neka_druga_funkcija ( tip1 p1, tip2 p2, ... ) {
    if ( "provjera p1" ) {
        "ili vrati grešku, ili prekini dretvu, ili ...";
    }
    ...
   x = malloc (...);
    if ( x == NULL ) {
        "prijavi grešku u neki dnevnik ili korisniku preko zaslona, ..."
        "ili vrati grešku, ili prekini dretvu, ili ...";
    }
    ...
}
```

3. [1 bod] Napraviti makro `POSTAVI(tip, podaci, duljina, poruka)` koji će popuniti varijablu strukture `struct poruka { short tip; char data[1]; }`. Pretpostaviti da je spremnički prostor odgovarajuće duljine za poruku već zauzet.

   Rj.

```
#define POSTAVI(tip, podaci, duljina, poruka)  \
do{ poruka.tip = tip; memcpy ( poruka.data, poruka, duljina ); }while(0)
```

4. [2 boda] Funkcije u nekoj datoteci koriste se i za jezgru i za programe (npr. `memset`). Međutim, obzirom na korištenje sklopovlja za pretvorbu logičkih adresa u fizičke za programe, dok jezgra koristi fizičke adrese (apsolutne), isti se kod dva puta prevodi i kasnije zajedno povezuje ('linka'). Stoga se pri prevođenju trebaju koristiti različita imena funkcija (npr. `memcpy` za programe te `kmemcpy` za jezgru). Napisati dio koda koji definira makro `FUNKCIJA(ime_funkcije)` koji treba koristiti pri deklaraciji funkcija. Preko makroa `JEZGRA` može se doznati je li funkciji treba dodati prefiks `k` (kada je `JEZGRA=1`) ili ne. Npr. korištenje makroa bi izgledalo:

   ```
   void * FUNKCIJA(memcpy) ( void *dest, const void *src, size_t num )
   ```

   što se u početnoj fazi prevođenja treba prevesti u:

   ```
   void * memcpy ( void *dest, const void *src, size_t num )
   ```

   ili

   ```
   void * kmemcpy ( void *dest, const void *src, size_t num )
   ```

   Rj.

   ```
   #if JEZGRA == 1
       #define FUNKCIJA(ime_funkcije)   k ## ime_funkcije
   #else
       #define FUNKCIJA(ime_funkcije)   ime_funkcije
   #endif
   ```

5. [1 bod] Navesti primjere gdje jedan program (dretva) zbog greške može narušiti cijeli sustav. Kojim mehanizmima se navedeni problemi mogu lokalizirati (da greške ne utječu na ostatak sustava)?

   Rj.
   Zbog mijenjanja podataka drugih procesa ili jezgre OS-a. Rješenje: koristiti upravljanje spremnikom koje će onemogućiti procesu da piše van ograđenog prostora.

   Zbog instrukcija koje upravljaju nekim dijelovima sustava. Primjerice, program može zabraniti prekide i time onemogućiti prihvat prekida svih naprava (a time i sata koji bi tu dretvu maknuo s procesora). Rješenje: procese izvoditi u korisničkom načinu rada u kojem ne mogu pokretati takve instrukcije.

6. [2 boda] U pseudokodu prikazati ostvarenje prekidnog podsustava. Ostvariti sve neophodne funkcije za korištenje iz jezgre. Rješenje mora biti potpunije, npr. nije dovoljno napisati 'spremni kontekst dretve' već 'spremni kontekst dretve u opisnik dretve'. Pretpostaviti da se pri prihvatu prekida automatski na stog pohranjuje programsko brojilo i identifikator prekida (broj) te da se u programsko brojilo upisuje vrijednost 10. Pretpostaviti da za svaki broj (do `BR_PREKIDA`) postoji samo jedan mogući izvor prekida.

   Rj.

```
//struktura podataka:
polje_kazaljki_na_funkciju obrada[BR_PREKIDA];

//sučelja za jezgru
inicijalizacija_prekidnog_podsustava() {
    za i = 0 do BR_PREKIDA-1
        obrada[i] = NULL;
    omogući prekidanje;
}

registriraj_prekid ( id, funkcija ) {
    ako je ( id > 0 && id <= BR_PREKIDA )
        obrada[id] = funkcija;
}

//obrada prekida
10: //na adresi 10
    pohrani sve korisničke registre procesora na stog;
    pozovi "obrada_prekida"
    obnovi sve korisničke registre procesora sa stoga;
    vrati se iz prekida; //učitaj PC, makni id sa stoga, dozvoli
       prekidanje

obrada_prekida() {
    x = dohvati_opisnik_aktivne_dretve ();
    kopiraj_kontekst_u_opisnik_dretve;
    id = dohvati_id_prekida_sa_stoga;
    ako je ( obrada[id] != NULL )
        obrada[id] (id);
    x = dohvati_opisnik_aktivne_dretve ();
    kopiraj_kontekst_iz_opisnika_dretve_x_na_stog;
}
```

7. [2 boda] Neki sklop detektira otkucaje sata te preko prikladnog sučelja zapisuje broj 1 na adresu `BEAT`. Korištenjem tog podatka ostvariti sustav koji će na zaslonu uređaja prikazivati:
   a) ukupan broj otkucaja (`b`)
   b) trenutnu frekvenciju otkucaja (broj otkucaja u minuti `bpm`)
   c) procjenjenu potrošnju kalorija po minuti (računati preko `cpm=fun1(bpm)`, `fun1` postoji)
   d) procjenjeni ukupan broj potrošenih kalorija (`cals=fun2(b)`, `fun2` postoji).

   Sustav posjeduje 16-bitovno brojilo koje odbrojava frekvencijom FREQ. Brojilo nije moguće promijeniti (resetirati) niti ono izaziva prekide, već nastavlja s nulom nakon što dosegne najveću vrijednost. Pretpostaviti da se ispis vrijednosti na zaslon zbiva jednostavnim upisom odgovarajućih vrijednosti na zasebne lokacije u spremniku (adrese: `B`, `BPM`, `CPM`, `CALS`). Broj otkucaja po minuti izračunavati na osnovu zadnja četiri otkucaja: $bpm = 60 \cdot 3/(t_4 - t_1)$. Upravljanje ostvariti upravljačkom petljom (beskonačnom petljom).

   Rj.

```
//struktura podataka:
#define MAXCNT  (1<<16)
int b, bpm;
int *pb = B, *pbpm = BPM, *pcpm = CPM, *pcals = CALS;
int *beat = BEAT, *brojilo = BROJILO;
int t[4] = {0,0,0,0}, T;

//upravljačka petlja
void upravljanje () {
    while (1) {
        if ( *beat ) {
            *beat = 0;
            b++;

            t[0] = t[1]; t[1] = t[2]; t[2] = t[3];
            t[3] = *brojilo;

            T = 0;
            if ( t[1] > t[0] )
                T += t[1] - t[0];
            else
                T += t[1] + MAXCNT - t[0];

            if ( t[2] > t[1] )
                T += t[2] - t[1];
            else
                T += t[2] + MAXCNT - t[1];

            if ( t[3] > t[2] )
                T += t[3] - t[2];
            else
                T += t[3] + MAXCNT - t[1];

            bpm = 60 * 3 * FREQ / T;

            *pb = b;
            *pbpm = bpm;
            *pcpm = fun1(bpm);
            *pcals = fun2(b);
        }
    }
}
```

8. [4 boda] U nekom računalnom sustavu postoji 64-bitovno brojilo koje odbrojava frekvencijom od 1 GHz. Korištenjem tog brojila ostvariti:
   a) praćenje dodijeljenog procesorskog vremena pojedinoj dretvi
   b) raspoređivanje podjelom vremena.

   Pretpostaviti da svaki poziv jezgrine funkcije započinje operacijom `deaktiviraj_dretvu` iz koje se poziva funkcija `ažuriraj_vremena` koju treba ostvariti (za a) dio).
   Nadalje, pretpostaviti da u sustavu postoji i drugi satni mehanizam koji periodički izaziva prekide (dovoljno velikom, ali nepoznatom i nestalnom frekvencijom). Iz tih se funkcija poziva `raspored ivanje_podjelom_vremena` koju treba ostvariti (za b) dio). Neka sve dretve trebaju dobiti kvant vremena `T`. Za samo raspoređivanje ne koristiti dodatne alarme već samo poziv `raspoređivanje _podjelom_vremena` koji je ugrađen u sve potrebne funkcije. Za praćenje dobivenog vremena po dretvi koristiti podatke prikupljene u a) dijelu zadatka. Zbog poziva iz drugih funkcija poneka će dretve dobiti i više vremena od T, ali manje ne smije. Za upravljanje dretvama koristiti pozive: `stavi_ u_pripravne(dretva)`, `prva=uzmi_prvu_pripravnu()`, `aktivna=dohvati_aktivnu()` te `postavi_aktivnu(dretva)`.
   Proširiti opisnike dretve po potrebi. Korištenje vremena pojednostaviti (npr. koristiti vrijeme u jedinicama nanosekunde u 64-bitovnim varijablama).

   Rj. a)
   1) uz pretpostavku da jezgrine funkcije traju kratko (zanemarivo).

```
#define BROJILO neka_adresa //neka brojilo odbrojava prema većim gore
#define FREQ    1000000000
#define MAX 0xffffffffffffffff  //2^64-1
long zadnje_očitanje_brojila = 0;
long *brojilo = BROJILO;

ažuriraj_vremena () {
    aktivna = dohvati_aktivnu();
    if ( *brojilo > zadnje_očitanje_brojila )
        t = *brojilo - zadnje_očitanje_brojila;
    else
        t = MAX - zadnje_očitanje_brojila + 1 + *brojilo;

    zadnje_očitanje_brojila = *brojilo;
    aktivna->dobiveno_vremena += t;
}
```

   2) uz pretpostavku da jezgrine funkcije traju duže (nezanemarivo) (informativno)

```
#define BROJILO neka_adresa //neka brojilo odbrojava prema gore
#define FREQ    1000000000
#define MAX 0xffffffffffffffff  //2^64-1
long *brojilo = BROJILO;

ažuriraj_vremena () {
    aktivna = dohvati_aktivnu();
    if ( *brojilo > aktivna->zadnje_očitanje_brojila )
        t = *brojilo - aktivna->zadnje_očitanje_brojila;
    else
        t = MAX - aktivna->zadnje_očitanje_brojila + 1 + *brojilo;

    aktivna->dobiveno_vremena += T;
}
prije_povratka_u_dretvu (aktivna) {
    aktivna->zadnje_očitanje_brojila = *brojilo;
}
```

Rj. b)

```
raspoređivanje_podjelom_vremena () {
    ažuriraj_vremena ();
    aktivna = dohvati_aktivnu();
    t = aktivna->dobiveno_vremena - aktivna->pocetak_kvanta;
    if ( t >= T ) {
        stavi_u_pripravne ( aktivna );
        aktivna = uzmi_prvu_pripravnu ();
        postavi_aktivnu ( aktivna );
        aktivna->pocetak_kvanta = aktivna->dobiveno_vremena;
    }
}
```

9. [3 boda] Neki ugrađeni sustav sadrži sljedeće spremnike:
   a) ROM na adresi 0x100000
   b) RAM na adresi 0x200000
   c) priručni spremnik na adresi 0xF000000.

   Programska potpora sastoji se od jezgre OS-a (u direktoriju `jezgra`) te programa P_1, P_2, ... P_6 koji se nalaze u direktorijima `programi/p1`, `programi/p2`, `...programi/p6`. Slika sustava će se posebnim alatima spremiti u ROM. Pri pokretanju sustava posebnim sklopovljem sve će se kopirati u RAM. Potom će tek započeti s radom izgrađena programska komponenta. Jezgrine funkcije i podaci jezgre najprije će se kopirati u priručni spremnik (i tamo ostati i od tamo koristiti). Programi P_1, P_2 i P_3 pokretati će se iz RAM-a (za njega ih treba pripremiti), dok će ostali iz priručnog spremnika, kamo će se kopirati pri pokretanju. Programi P_4, P_5 i P_6 pokreću se pojedinačno – nikad nisu dva istovremeno aktivna (a i ne bi stalo više od jednog u priručni spremnik, uz jezgru). Dakle, programe P_4, P_5 i P_6 pripremiti za istu početnu adresu u priručnom spremniku (odmah iza jezgre). Sustav nema sklopovlje za dinamičko pretvaranje adresa – sve adrese moraju biti pripremljene za lokacije s kojih će se koristiti. Napisati skripte za povezivača.

Rj.

```
ROM = 0x100000; RAM = 0x200000; CACHE = 0xF000000;
SECTIONS {
    kernel_1st_load = RAM;
    kernel_copy_to = CACHE;
    .kernel kernel_copy_to : AT ( kernel_1st_load ) {
        jezgra* ( * )
    }
    p123_1st_load = RAM + SIZEOF ( .kernel );
    .p123 p123_1st_load : AT ( p123_1st_load ) {
        programi/p1* ( * )
        programi/p2* ( * )
        programi/p3* ( * )
    }
    p456_copy_to = kernel_copy_to + SIZEOF ( .kernel );
    p4_1st_load = p123_1st_load + SIZEOF ( .p123 );
    .p4 p456_copy_to : AT ( p4_1st_load ) {
        programi/p4* ( * )
    }
    p5_1st_load = p4_1st_load + SIZEOF ( .p4 );
    .p5 p456_copy_to : AT ( p5_1st_load ) {
        programi/p5* ( * )
    }
    p6_1st_load = p5_1st_load + SIZEOF ( .p5 );
    .p6 p456_copy_to : AT ( p6_1st_load ) {
        programi/p6* ( * )
    }
}
```

10. [3 boda] U sustavu koji koristi sklopovsku potporu za dinamičko upravljanje spremnikom, programi koriste logičke adrese dok u se u jezgrinim funkcijama koriste fizičke (apsolutne). Ostvariti jezgrinu funkciju `j_najveća` koja za zadana imena datoteka vraća ime i veličinu najveće datoteke (od zadanih imena). Neka je funkcija koja se poziva iz programa:

```
int najveća ( char **imena, size_t *veličina, char **najveća ) {
    izazovi_programski_prekid;
}
```

Iz obrade programskog prekida poziva se jezgrina funkcija: `j_najveća(void *parametri)` gdje je jedini parametar adresa (fizička) stoga pozivajuće dretve (adresa gdje se nalazi prvi parametar `imena`). Izvođenje jezgrine funkcije obavlja se korištenjem fizičkog načina adresiranja (pretvaranje adresa je isključeno). Uz pretpostavku da postoji pomoćna funkcija `vrati_veličinu(ime)` koja vraća veličinu zadane datoteke, ostvariti jezgrinu funkciju `j_najveća`. Zanemariti povratnu vrijednost funkcije (ono što `najveća` vraća kao povratnu vrijednost, ne preko parametara). Adresu početka spremničkog prostora trenutnog procesa može se dohvatiti sa `početna_adresa_procesa (NULL)`.

Rj.

```
int j_najveća ( void *parametri ) {
    char **imena;
    size_t *veličina;
    char **najveća;

    imena = *((char ***) parametri); parametri += sizeof (char **);
    veličina = *((size_t **) parametri); parametri += sizeof (size_t *);
    najveća = *((char ***) parametri);

    //adrese su u logičkom obliku, pretvori ih u apsolutni
    početna = početna_adresa_procesa ( NULL );
    imena += početna;
    veličina += početna;
    najveća += početna;

    max = -1;
    imax = 0;
    for ( i = 0; imena[i] != NULL; i++ ) {
        ime = imena[i] + početna; //pretvori u fiz. adresu
        vel = vrati_veličinu ( ime );
        if ( vel > max ) {
            imax = i;
            max = vel;
        }
    }
    *veličina = max;
    *najveća = imena[imax]; //ovo već je u logičkim adresama
}
```

1. [2 boda] Primjerima (i dodatnim opisom) demonstrirati smisao korištenja ključnih riječi `static`, `volatile` i `extern`.

2. [2 boda] Napisati makro `M1(A,B,C)` koji će vratiti vrijednost (`A`, `B` ili `C`) za koju funkcija `F1(x)` (`x` je `A`, `B` ili `C`) daje najveću vrijednost. Pretpostaviti jednostavne parametre (koji ne mijenjaju ništa, npr. neće biti `a++` ili `fun(x)` kao parametri, ali može biti `x+y`).

   Rješenje:

```
#define M1(A,B,C) \
( F1(A)>F1(B)? (F1(A)>F1(C)?(A):(C)) : ( (F1(B)>F1(C)?(B):(C)) ) )
```

3. [4 boda] Izvorni kod nekog sustava sastoji se od nekoliko datoteka raspoređenih u direktorije `jezgra`, `ui` te `programi`. U svakom se direktoriju nalaze tri datoteke s izvornim kodom čije se ime sastoji od imena direktorija te dodatkom redna broja (npr. u `ui` se nalaze `ui1.c`, `ui2.c` i `ui3.c`). Napisati datoteke `Makefile` koje treba staviti u svaki od direktorija (koje se brinu za prevođenje datoteka u tim direktorijima) uz jedan `Makefile` u početnom direktoriju. Izlazna datoteka treba se zvati `program`. Pretpostaviti da za prevođenje i povezivanje ne trebaju nikakve posebne zastavice. (Iz jednog `Makefile`-a se drugi, u nižem direktoriju `x`, može pozvati s `make -C x`.)

   Rješenje:

   jezgra/Makefile:

```
jezgra.o: jezgra1.o jezgra2.o jezgra3.o
    (LD) -r jezgra1.o jezgra2.o jezgra3.o -o jezgra.o
#ovo dalje nije neophodno, implicitna pravila su dovoljna
jezgra1.o: jezgra1.c
    (CC) -c jezgra1.c
jezgra2.o: jezgra2.c
    (CC) -c jezgra2.c
jezgra3.o: jezgra3.c
    (CC) -c jezgra3.c
```

   ui/Makefile:

```
ui.o: ui1.o ui2.o ui3.o
    (LD) -r ui1.o ui2.o ui3.o -o ui.o
```

   programi/Makefile:

```
programi.o: programi1.o programi2.o programi3.o
    (LD) -r programi1.o programi2.o programi3.o -o programi.o
```

   Makefile:

```
program: jezgra/jezgra.o ui/ui.o programi/programi.o
    (LD) jezgra/jezgra.o ui/ui.o program/iprogrami.o -o program
jezgra/jezgra.o:
    (MAKE) -C jezgra
ui/ui.o:
    (MAKE) -C ui
programi/programi.o:
    (MAKE) -C programi
```

   Može i drukčije, npr.:

   jezgra/Makefile:

```
jezgra1.o: jezgra1.c
jezgra2.o: jezgra2.c
jezgra3.o: jezgra3.c
```

ui/Makefile

```
ui1.o: ui1.c
ui2.o: ui2.c
ui3.o: ui3.c
```

programi/Makefile

```
programi1.o: programi1.c
programi2.o: programi2.c
programi3.o: programi3.c
```

Makefile:

```
JEZGRA = jezgra/jezgra1.o jezgra/jezgra2.o jezgra/jezgra3.o
UI = ui/ui1.o ui/ui2.o ui/ui3.o
PROGRAMI= programi/programi1.o programi/programi2.o programi/programi3.o

program: $(JEZGRA) $(UI) $(PROGRAMI)
    (LD) $(JEZGRA) $(UI) $(PROGRAMI) -o program
$(JEZGRA):
    (MAKE) -C jezgra
$(UI):
    (MAKE) -C ui
$(PROGRAMI):
    (MAKE) -C programi
```

4. [4 boda] Za neki ugradbeni sustav zadani su zahtjevi na pripremu programa za učitavanje u ROM. Program se sastoji od dvije datoteke `d1.c` i `d2.c`. U prvoj `d1.c` nalazi se (pored ostalog) i polje `int a[N]={/*početne vrijednosti*/}` koje treba pripremiti za učitavanje (za rad) na adresi `A1`. U drugoj datoteci nalazi se slična struktura `float b[]={/*početne vrijednosti */}` koju treba pripremiti za adresu `B1`. Kopiranje polja `a` i `b` iz ROM-a na zadane adrese (`A1`, `B1`) treba napraviti u funkciji `move_ab()`. Napisati skriptu za povezivača, deklaracije polja `a` i `b` (proširiti već navedeno) te funkciju `move_ab()`. Adresa ROM-a je `R1`, adresa RAM-a `M1`. Pretpostaviti da ostala kopiranja potrebnih dijelova iz ROM u RAM radi neka druga funkcija (nije ju potrebno ostvariti) te da sve instrukcije i konstante ostaju u ROM-u, a sve varijable se kopiraju u RAM, na početak.

Rješenje:

ldscript.ld

```
SECTIONS{
  .rom R1 : AT(R1) {
    * (.text .rodata)
  }
  m_poc = R1 + SIZEOF(.rom);
  .ram M1 : AT(m_poc) {
    * (.data .bss)
  }
  a_poc = m_poc + SIZEOF(.ram);
  .v_a A1 : AT (a_poc) {
    d1.o ( .v_a )
  }
  b_poc = a_poc + SIZEOF(.b_poc);
  .v_b B1 : AT (b_poc) {
    d2.o ( .v_b )
  }
  kraj = b_poc + SIZEOF(.v_b);
}
```

kodovi

```
/* u d1.c */
int a[N]={/* poč. vr. */}__
attribute__((section(".v_a"));

/* u d2.c */
float b[M]={/* poč. vr. */}__
attribute__((section(".v_b"));

/* "negdje" */
void move_ab(){
  extern char a_poc, b_poc, kraj;
  char *a = &a_poc, *b = &b_poc;
  char *am = A1, *bm = B1;
  while ( a < b )
    *am++ = *a++;
  while ( b < &kraj )
    *bm++ = *b++;
}
```

5. [4 boda] Sklop za prihvat prekida ima dva registra: KZ (kopija zastavica) i TP (tekući prioritet). Bitovi različiti od nule u registru KZ označavaju da dotična naprava traži obradu prekida, dok nule označavaju naprave koje ne traže prekid ili je njihov zahjev prihvaćen (u obradi). Naprava spušta svoj zahtjev kada se pozove funkcija za obradu prekida te naprave. U sustavu ima N naprava i svaka je spojena na svoj ulaz sklopa za prihvat prekida (svaka ima različiti prioritet). Prekidi većeg prioriteta trebaju prekidati obradu prekida manjeg prioriteta. U registru TP bitovi postavljeni u 1 označavaju da je dotični prekid u obradi (procesor postavlja i briše registar TP – to treba ugraditi u kod). Npr. ako je vrijednost KZ $= 00010011_2$ i TP $= 00100100_2$ tada naprave s indeksima/prioritetima 4, 1 i 0 imaju postavljen zahtjev za prekid dok se trenutno obrađuje zahtjev naprave 5, a prekinuta je obrada naprave 2 (koja se treba nastaviti po završetku prioritetnijih). Neka postoji funkcija $msb(x)$ koja vraća indeks najznačajnije jedinice (npr. $msb(001000_2) = 3$). Ostvariti prekidni podsustav (`void inicijaliziraj()`, `void registriraj_prekid (int irq, void (*obrada)())` te `void prihvat_prekida()` koja se poziva svaki puta kad se prekid prihvati, bez argumenata!) za opisani sustav uz pretpostavku da će sklop proslijediti zahtjev većeg prioriteta od tekućeg prema procesoru, dok će one manjeg prioriteta zadržati. Pri prihvatu prekida, prije poziva `prihvat_prekida` kontekst prekinuta posla spremljen je na stog, a nakon povratka iz iste funkcije sa stoga se obnavlja kontest (ne treba ga programski spremati/obnavljati).

Rješenje:

```
void (*fun[N])();
void inicijaliziraj () {
    int i;
    for ( i = 0; i < N; i++ )
        fun[i] = NULL;
    TP = 0;
}
void registriraj_prekid ( int irq, void (*obrada)() ){
    fun[irq] = obrada;
}
void prihvat_prekida () {
    int irq = msb ( KZ );
    TP = TP | (1<<irq);
    dozvoli_prekidanje();
    fun[irq]();
    zabrani_prekidanje();
    TP = TP ^ (1<<irq);
}
```

6. [4 boda] Neki sustav posjeduje 10 bitovno brojilo na adresi `CNT` koje odbrojava frekvencijom od 100 kHz. Upisom neke vrijednosti u brojilo započinje odbrojavanje prema nuli. Kada brojilo dođe do nule, izaziva prekid te se učitava zadnja upisana vrijednost pa ponovno kreće s odbrojavanjem. Izgraditi sustav upravljanja vremenom koji treba imati sučelja:
   a) inicijalizacija podsustava: `void inicijaliziraj()`
   b) obrada prekida brojila: `void prekid_sata()`
   c) dohvat trenutna sata: `long dohvati_vrijeme()` (vraća vrijeme u ms)
   d) promjena trenutna sata: `void postavi_vrijeme(long novo_vrijeme_ms)`
   e) postavljanje alarma: `void alarm(long za_koliko_ms, void (*obrada)())`

   Sučelje koristi vrijednost sata u milisekundama (pretpostaviti da je tip `long` dovoljan za prikaz sata u milisekundama). Međutim, interno, obzirom da zahtjevi za postavljanjem alarma mogu doći u bilo kojem trenutku, preciznost treba biti veća (u rezoluciji brojila). Promjena sata i postavljanje alarma briše prethodno postavljeni alarm (on se ne poziva).

Rješenje:

```
#define MAX 1000 //=> 10 ms

long sat_ms, sat_us; // sat u ms i ostatak u mikrosekundama
long odgoda; //u ms
int ucitano; //100, 200, ... 1000
void (*funkcija)();
short *br = CNT;

void inicijaliziraj () {
    sat_ms = sat_us = 0;
    odgoda = 0;
    ucitano = MAX;
    *br = ucitano;
}
long dohvati_vrijeme () {
    return sat_ms + ( sat_us + (ucitano - *br) * 10 ) / 1000;
}
void postavi_vrijeme ( long novo_vrijeme_ms ) {
    sat_ms = novo_vrijeme_ms;
    sat_us = 0;
    odgoda = 0;
    *br = ucitano = MAX;
}
void alarm ( long za_koliko_ms, void (*obrada)()) {
    odgoda = za_koliko_ms;
    funkcija = obrada;
    ucitano = odgoda * 100;
    if ( ucitano > MAX )
        ucitano = MAX;
    *br = ucitano;
}
void prekid_sata() {
    sat_us += ucitano * 10;
    sat_ms += sat_us / 1000;
    sat_us = sat_us % 1000;

    if ( odgoda > 0 ) { // nije još
        odgoda -= ucitano / 100;
        if ( odgoda > 0 ) {
            ucitano = odgoda * 100;
            if ( ucitano > MAX )
                ucitano = MAX;
            else
                *br = ucitano;
        }
        else { // alarm
            odgoda = 0;
            if ( ucitano < MAX )
                *br = ucitano = MAX;
            funkcija();
        }
    }
}
```

1. (2) Sljedeći makro za zbrajanje dva kompleksna broja ima nekoliko nedostataka:
   `#define CADD(A,B,Z) {Z->x = A->x + B->x; Z->y = A->y + B->y;}`
   Popraviti makro tako da se ti nedostaci uklone. Makro ne vraća vrijednost, ali se treba moći pozvati od bilo kuda i sa složenim parametrima (npr. i poziv `CADD(a=xz(x,y),z++,&w)` treba raditi ispravno).

```
3 stvari su potrebne:
1. do-while
2. zagrade oko A/B/Z
3. lokalne varijable da se svaki argument samo jednom javlja u kodu

#define CADD(A,B,Z)                    \
do {                                   \
    typeof(A) tmp_a = (A);             \
    typeof(B) tmp_b = (B);             \
    typeof(Z) tmp_z = (Z);             \
    tmp_z->x = tmp_a->x + tmp_b->x;    \
    tmp_z->y = tmp_a->y + tmp_b->y;    \
}                                      \
while(0)

typeof(A) daje tip argumenta
npr. za A tipa "struct z *" dio
    typeof(A) tmp_a = (A);
će se preprocesirati u:
    struct z * tmp_a = (A);
```

2. (2) Pero i Ana koriste git za rad na zajedničkom projektu. U nekom trenutku oboje su krenuli s identičnim repozitorijem, ali su tada oboje radili različite promjene nad istom datotekom `modul.c`. Ana je prva dovršila svoj posao i postavila svoje promjene u zajednički repozitorij.
   a) Kako će sustav reagirati kad Pero bude htio svoje promjene postaviti u repozitorij?
   b) Što će morati Pero napraviti da i svoje promjene stavi u repozitorij?

```
a) neće mu dati jer ne sadrži Anine izmjene
   sugerirati će mu da napravi "pull"

b) 1. napraviti pull
   2. ako ima konflikata (a vjerojatno ima) treba ih riješiti
   3. add+commit+push (ili slično)
```

3. (4) Izvorni kod nekog sustava sastoji se od datoteka *arch.c*, *kernel.c* i *programs.c* u istoimenim direktorijima *arch*, *kernel* i *programs*. Prevođenje datoteka je različito u različitim direktorijima te treba napraviti zasebne upute – *Makefile*-ove, zasebni za svaki od navedenih direktorija te jedan u početnom direktoriju koji prvo mora aktivirati navedene s `$(make)-C dir` te na kraju povezati sve objekte u sliku sustava `slika.elf`. Napišite sadržaj potrebnih *Makefile*-ova. Zastavice za prevođenje datoteka u pojedinom direktoriju označiti sa: `zast_<dir>` (ostale zastavice nisu potrebne).

`arch/Makefile:`

```
CFLAGS = zast_arch
arch.o: arch.c #+implicitna pravila

ili
arch.o: arch.c
    (CC) zast_arch -o arch.o -c arch.c
```

`slično za kernel/Makefile i programs/Makefile`

`glavni Makefile:`

```
OBJEKTI = arch/arch.o kernel/kernel.o programs/programs.o
slika.elf: $(OBJEKTI)
    $(CC) $(LDFLAGS) -o $@ $^ #ili sve navesti izravno

.PHONY: $(OBJEKTI) #nije neophodno
$(OBJEKTI): #ili tri ovakva zasebna za: arch/arch.o kern...
    $(MAKE) -C $(dir $@)
```

4. (4) Izvorni kod nekog sustava sastoji se od datoteka u direktorijima *arch*, *kernel* i *programs*. Pokretač sustava (bootloader) koji se nalazi u drugom dijelu ROM-a (koji se ne može mijenjati) pri pokretanju sustava će dio iz ROM-a s adresa 0x10000 do 0x20000 kopirati u priručni spremnik na adresi 0x60000 – u taj dio treba učitati sve što nastaje iz datoteka iz direktorija *arch* i *kernel*. Ostatak sadržaja ROM-a, od 0x20000 do 0x50000, pokretač će kopirati u radni spremnik na adresu 0x100000 – u taj dio treba učitati sve što nastaje iz direktorija *programs*. Napisati skriptu za povezivanje koja će pripremiti sliku sustava koju treba upisati u ROM (posebnim programom koji će znati interpretirati sliku), ali da se pripreme za izvođenje iz priručnog spremnika, odnosno radnog spremnika, prema opisanoj specifikaciji.

`ldscript.ld:`

```
SECTIONS {
    .cache 0x60000 : AT ( 0x10000 ) {
        arch* (*)
        kernel* (*)
    }
    .ram 0x100000 : AT ( 0x20000 ) {
        programs* (*)
    }
}
```

5. (4) Neki sklop za prihvat prekida ima 16 ulaza. Na svaki ulaz može biti spojena samo jedna naprava. Za svaki ulaz postoji upravljački/statusni registar (`PP[i]`). Postavljanjem jedinice u bit 0 tog registra omogućava se proslijeđivanje prekida prema procesoru (nulom se to onemogućava). Bit 1 će postaviti naprava spojena na taj ulaz kada zahtijeva prekid. Ostali bitovi od `PP[i]` se ne koriste. Pretpostaviti da svi ulazi imaju jednak prioritet te da je vjerojatnost preklapanja dva različita zahtjeva zanemariva (uključujući obradu). Ostvariti prekidni podsustav, tj. funkcije: `void irq_init(),void *irq_register(int irq, void (*handler)(int)), int irq_enable(int irq),int irq_disable(int irq),void irq_handler()` (ova zadnja se prva poziva pri prihvatu svakog prekida). Funkcija `irq_register` vraća prethodno registriranu funkciju, dok `irq_enable` i `irq_disable` vraćaju prethodno stanje prihvata prekida za taj broj.

```
#define IRQS 16
static void (*ih[IRQS])(int);

void irq_init () {
    int i;
    for ( i = 0; i < IRQS; i++ ) {
        ih[i] =  NULL;
        PP[i] = 0; //zabrani prekide
    }
}
void *irq_register ( int irq, void (*handler)(int) ) {
    if ( irq < 0 || irq >= IRQS )
        return NULL;
    void *old_hanlder = ih[irq];
    ih[irq] = handler;
    PP[irq] |= 1; //možda zahtjev za prekid već čeka
}
int irq_enable ( int irq ) {
    if ( irq < 0 || irq >= IRQS )
        return -1;
    int prev = PP[irq] & 1;
    PP[irq] |= 1; //možda zahtjev za prekid već čeka
    return prev;
}
int irq_disable ( int irq ) {
    if ( irq < 0 || irq >= IRQS )
        return -1;
    int prev = PP[irq] & 1;
    PP[irq] &= ~1;
    return prev;
}
void irq_handler () {
    int i;
    for ( i = 0; i < IRQS; i++ ) {
        if ( PP[i] & 3 != 0 ) { //omogućeni prekidi i postoji zahtjev
            ih[i](i);
            PP[i] &= ~2;
        }
    }
}
```

6. (4) Neko brojilo odbrojava frekvencijom od 10 MHz i kada dođe do nule izaziva prekid. Trenutna vrijednost brojila se može očitati i postaviti preko registra `BR`. Ostvariti podsustav za upravljanje vremenom koje ostvaruje sat i jedan alarm s internom preciznošću od 0,1 μs sa sučeljem:
`void clock_init();void clock_set(u64 clock);u64 clock_get();`
`void clock_alarm(u64 delay, void (*handler)());void clock_irqhandler()`
Pretpostaviti da su brojilo i procesorska riječ 64 bitovni podaci `u64` (sve vrijednosti povezane s vremenom stanu u njih u traženoj preciznosti), a sat se prema van iskazuje u mikrosekundama (parametri `clock`, `delay`). Prekide izazivati samo kada je to neophodno – kada je postavljen alarm (inače se neće dogoditi uz maksimalno veliku vrijednosti u BR).

```
#define MAXCOUNT 0xFFFFFFFFFFFFFFFF

static u64 time;         //vrijeme u desetinkama mikrosekundi
static u64 last_load;    //zadnja učitana vrijednost u brojilo
static void (*alarm)(); //funkcija za aktivaciju alarma

void clock_init () {
    time = 0;
    BR = last_load = MAXCOUNT;
}

u64 clock_get () {
    return ( time + last_load - BR ) / 10;
}

void clock_set ( u64 clock ) {
    time = clock * 10;
    BR = last_load = MAXCOUNT; //mičem alarm
}

void clock_alarm ( u64 delay, void (*handler)() ) {
    time += last_load - BR;
    BR = last_load = delay * 10;
    alarm = handler;
}

void clock_irqhandler() {
    //ovdje će doći samo ako je alarm bio postavljen
    time += last_load;
    BR = last_load = MAXCOUNT;
    alarm();
}
```

Pisati čitko – nečitak odgovor ne donosi bodove.

1. (2) Napisati makro `FUNKCIJA(x,y,z)` koji će za tri ulazne vrijednosti x, y i z vratiti vrijednost funkcije: $f(x, y, z) = x + y \cdot z$.

```
- samo dodati zagrade oko svega, pojedinačno i zajedno

#define FUNKCIJA(x,y,z) ( (x) + (y) * (z) )

- obzirom da se svaki argument koristi samo jednom nije potrebno
   dodavati typeof(x) _x_ = (x) i slično
- krivo je koristiti do-while(0) !!!
```

2. (2) Napisati makro `SREDNJA(x,y,z)` koji će za tri ulazne vrijednosti x, y i z vratiti onu ulaznu vrijednost koja nije ni najmanja ni najveća. Pretpostaviti da se taj makro može pozivati i sa složenim argumentima (npr. `SREDNJA(i++, b=fun1(c), fun2())`).

```
#define SREDNJA(x,y,z)                                                  \
({                                                                      \
    typeof (x) _x_ = (x);                                               \
    typeof (y) _y_ = (y);                                               \
    typeof (z) _z_ = (z);                                               \
                                                                        \
    (_x_ > _y_)?( _y_ > _z_ ? _y_ : ( _x_ > _z_ ? _z_ : _x_ ) ) : \
                ( _x_ > _z_ ? _x_ : ( _y_ > _z_ ? _z_ : _y_ ) )    \
})

- ovdje je potrebno koristiti typeof
  (jer se puno puta koristi isti argument)
- krivo je koristiti do-while(0)
- krivo je koristiti "return"
```

3. Student svoj zadatak za laboratorijsku vježbu predaje preko git repozitorija. Prvu vježbu je predao ("commit" i "push") i započeo je rad na drugoj (samo "commit"). Nastavnik je pregledao prvu vježbu i stavio svoje komentare u repozitorij (promjenio neke datoteke te napravio i "commit" i "push"). Što mora student napraviti da bi nastavio rad na drugoj vježbi i onda i to objavio:

   a) (2) ako su datoteke za drugu vježbu u drugoj mapi (npr. u lab2) ili

   b) (2) ako su datoteke za drugu vježbu iste kao i za prvu (prva se nadograđuje)?

```
a) prije "push" treba napraviti "pull" (on radi "automatski merge" i
   nema konflikata)
b) git pull + pogledati konflikte i popraviti ih, onda add/commit/push
```

4. (2) Dinamičko upravljanje spremnikom u nekom je sustavu ostvareno korištenjem metode "prvi odgovarajući" (npr. kao u Benu). Navesti sve razloge "neproduktivnog" korištenja spremnika zbog toga, tj. na što se sve troši/gubi spremnički prostor, osim za pohranjivanje podataka potrebnih programu.

```
- zaglavlja za blokove
- poravnanje na veličinu zaglavlja
- dodjela cijelog bloka kad je ostatak premali
- fragmentacija
```

5. (3) Radi upravljanja nekim sustavom treba generirati prekid **svakih** 100 mikrosekundi i pozvati funkciju `ažuriraj()`. Također, **jednom** unutar svakih 330 mikrosekundi (počevši od t=0) treba pozvati `proračunaj()` (obje funkcije postoje i njihovo izvođenje je kratko, do nekoliko mikrosekundi). Za to na raspolaganju stoji 16-bitovno brojilo (na adresi `CNT`) koje odbrojava frekvencijom od 25 MHz od zadnje učitane vrijednosti do nule, izaziva prekid i ponovno kreće s odbrojavanjem od iste vrijednosti. Ostvariti zadani sustav – funkciju `inicijalizacija()` koja se poziva na početku, funkciju `prekid_brojila()` koja se poziva na svaki prekid brojila, te opisati dodatno potrebnu strukturu podataka. Radi uštede energije prekid brojila se treba javljati što rijeđe moguće, ali opet da se zadovolje navedeni zahtjevi.

```
za prekid svakih 100 us, u brojilo treba staviti:
100 us / (1/25 MHz) = 2500 (stane u 16 bita!)

varijable:
long t; // vrijeme u us
long t330; // praćenje intervala od 330 us

void inicijalizacija() {
    t = t330 = 0;
    *CNT = 2500;
}

void prekid_brojila() {
    t += 100;
    ažuriraj();
    if ( t330 <= t ) {
        t330 += 330;
        proračunaj();
    }
}
```

6. Izvorni kod nekog sustava sastoji se od datoteka s izvornim kodom u C-u koje se nalaze u mapama (direktorijima) *boot*, *arch*, *kernel* i *progs*. Popis datoteka u svakom direktoriju (tj. objekata koji od njih nastaju) zadan je u zasebnim datotekama (npr. *arch/files.txt*) u obliku `dir_OBJS=dat1.o dat2.o ...` (`dir` je ime mape, `BOOT`, `ARCH` ...). Prilikom prevođenja datoteka potrebno je koristiti zastavice `Z1=5`, `Z2=1` i `Z3=15`. Dodatno, za prevođenje datoteka iz mapa *boot*, *arch* i *kernel* potrebno je postaviti zastavicu `KERNEL`. Uređaj za koji se sustav priprema ima ROM na adresi 0x10000 i RAM na adresi 0x100000. Ugrađeni program pokretač (boot loader, koji nije dio navedena izvorna koda) će odmah pri pokretanju kopirati sve iz ROM-a u RAM te započeti s izvođenjem koda na adresi 0x100000. Stoga treba program tako pripremiti da napočetku budu početne instrukcije (instrukcije jedine datoteke iz mape *boot*).

   a) (3) Napisati skriptu povezivača (*boot/ldscript.ld*).

   b) (4) Napisati *Makefile* (ili više njih, za svaki direktorij zaseban) za prevođnje sustava.

   Pri povezivanju, skripta povezivača se dodaje uz zastavicu `-T`:
   `ld -o ime_slike popis-objekata -T ime_skripte dodatne-zastavice`
   Koristiti recepte za prevođenje više datoteka u obliku:
   `mapa/%.o: mapa/%.c` (znak % mijenja više znakova)
   `$(CC)-c $< -o $@ <+dodatne zastavice>`.
   Za gornji primjer, varijabla `$<` bi sadržava samo ime datoteke (npr. `mapa/dat1.c`), a varijabla `$@` ime cilja (npr. `mapa/dat1.o`).

a) boot/ldscript.ld:

```
ENTRY(0x100000) /* nije nužno */
SECTIONS {
    .sve 0x100000 : AT ( 0x10000 ) {
        *boot* (.text)
        * (*)
    }
}
```

(može i drukčije)

b)
U tekstu se nespretno spominju "zastavice" umjesto "makroi" ili "vrijednosti" pa su i rješenja s -Z1=5 i slično ispravna iako je zamišljeno da se to dodaje sa zastavicom -D (-D Z1=5 -D Z2=1 itd.)

i) jedan Makefile u početnom direktoriju:

```
include boot/files.txt arch/files.txt kernel/files.txt progs/files.txt
CFLAGS1 = -D Z1=5 -D Z2=1 -D Z3=15 -D KERNEL
CFLAGS2 = -D Z1=5 -D Z2=1 -D Z3=15

bOBJS := $(addprefix boot/,$(boot_OBJS))
aOBJS := $(addprefix arch/,$(arch_OBJS))
kOBJS := $(addprefix kernel/,$(kernel_OBJS))
pOBJS := $(addprefix progs/,$(progs_OBJS))
# iako su ove transformacije potrebne, priznata su i rješenja bez njih

image.elf: $(bOBJS) $(aOBJS) $(kOBJS) $(pOBJS)
    ld -o $@ $^ -T boot/ldscript.ld

boot/%.o: boot/%.c
    gcc -c $< -o $@ $(CFLAGS1)
arch/%.o: arch/%.c
    gcc -c $< -o $@ $(CFLAGS1)
kernel/%.o:kernel/%.c
    gcc -c $< -o $@ $(CFLAGS1)
progs/%.o:progs/%.c
    gcc -c $< -o $@ $(CFLAGS2)
```

ii) po jedan Makefile u svakom direktoriju:

Makefile (početni):

```
image.elf: boot/boot.o arch/arch.o kernel/kernel.o progs/progs.o
    ld -o $@ $^ -T boot/ldscript.ld

%.o: # jedno pravilo za sve
    make -C $(dir $@)
```

```
# ili raspisati za svaki zasebno:
# boot/boot.o:
#     make -C boot
# ...
```

boot/Makefile:

```
include files.txt
CFLAGS = -D Z1=5 -D Z2=1 -D Z3=15 -D KERNEL

boot.o: $(boot_OBJS)
    gcc -r $^ -o $@

# implicitna pravila su dovoljna za izgradnju pojedinih .o datoteka
# ili koristiti recept:
# %.o: %.c
#     gcc -c $< -o $@ $(CFLAGS)
```

slično za arch i kernel;
za progs samo zadnju zastavicu maknuti (bez -D KERNEL)

Međuispit iz predmeta Operacijski sustavi za ugrađena računala
26. 4. 2022. Student: _____

Zadan je program u nekoliko datoteka (za zadatke 1-3):

fib.c
```
#include <stdlib.h>
#include <string.h>

static int zadnji = 1, predzadnji = 1;
static void ispisi();

int daj_max_fib_do_n (int n) {
    int iduci = zadnji + predzadnji;
    while (iduci < n) {
        predzadnji = zadnji;
        zadnji = iduci;
        iduci = zadnji + predzadnji;
    }
    ispisi();
    return zadnji;
}
static void ispisi() {
    int i;
    char buf[15];
    volatile char *serija = (char*) 0x1234;

    memset(buf, 0, 15);
    itoa(zadnji, buf, 10);
    for (i = 0; buf[i]; i++)
        *serija = buf[i];
}
```

sqrt.c
```
#include <stdlib.h>
#include <string.h>
#include <math.h>

static double zadnji = (A);
static double sqrt_zadnji = 215.3;

static void ispisi();

double daj_max_sqrt_do_n (int n) {
    double iduci = zadnji + 1;
    double sqrt_iduci = sqrt(iduci);
    while (sqrt_iduci < n) {
        zadnji = iduci;
        sqrt_zadnji = sqrt_iduci;
        iduci = zadnji + 1;
        sqrt_iduci = sqrt(iduci);
    }
    ispisi();
    return sqrt_zadnji;
}
static void ispisi() {
    int i;
    char buf[15];
    volatile char *serija = (char*) 0x123C;

    memset(buf, 0, 15);
    ftoa(zadnji, buf, 10);
    for (i = 0; buf[i]; i++)
        *serija = buf[i];
}
```

(samo **bold crveno** je neophodno za bodove)

prim.c
```
#include <stdlib.h>
#include <string.h>

static int zadnji = (A);
static void ispisi();

int daj_max_prim_do_n (int n) {
    int i, j;
    for (i = zadnji; i < n; i++) {
        for (j = 2; j < i/2; j++)
            if (i % j == 0) break;
        if (j == i/2) zadnji = i;
    }
    ispisi();
    return zadnji;
}
static void ispisi() {
    int i;
    char buf[15];
    volatile char *serija = (char*) 0x1238;

    memset(buf, 0, 15);
    itoa(zadnji, buf, 10);
    for (i = 0; buf[i]; i++)
        *serija = buf[i];
}
```

main.c
```
#include <time.h>
int daj_max_fib_do_n (int n);
int daj_max_prim_do_n (int n);
double daj_max_sqrt_do_n (int n);

//#define A 13 => preko Makefile-a za sve
#define B    (A) * 37
static void ispisi();
static int maxn;

int main() {
    int i, fib, prim;
    double sqrtn;
    struct timespec t;

    t.tv_sec = t.tv_nsec = 0;
    for (i = A; i < B + 1; i++) {
        fib = daj_max_fib_do_n (i);
        prim = daj_max_prim_do_n (i);
        sqrtn = daj_max_sqrt_do_n (i);
        if (fib > prim) maxn = fib;
        else maxn = prim;
        if ((int)sqrtn > maxn)
            maxn = (int)sqrtn;
        ispisi();
    }
    return 0;
}
static void ispisi() {
    volatile char *serija = (char*) 0x1230;
    *serija = '0' + maxn % 7;
}
void premjesti() {…}
```

1. (2) **Popraviti** zadani kod dodavanjem odgovarajućih ključnih riječi, … (riješiti na ovom papiru).

2. (4) Napisati pripadni **`Makefile`** prema uobičajenim pravilima uz korištenje uobičajenih implicitnih varijabli (`CFLAGS, LDFLAGS, LDLIBS`) i implicitna pravila za prevođenje (za povezivanje napisati potpune upute). U kodovima se koristi vrijednost `A` koju treba pomoću `Makefile`-a zamijeniti s vrijednošću zbroja varijabli okoline `BROJ1` i `BROJ2` (ako nisu postavljene, postaviti ih u `Makefile`-u na vrijednosti 7 i 77). Izlazni program neka se zove `program`. Neka se kod optimira obzirom na veličinu (zastavica `-Os`). Pri povezivanju koristiti skriptu za povezivanje (dodati `-T skripta.ld`, skriptu napisati u okviru slijedećeg zadatka).

```
Makefile
BROJ1 ?= 7      # može i bez ?
BROJ2 ?= 77     # može i bez ?
CFLAGS = -D A=($(BROJ1)+$(BROJ2))
LDFLAGS = -Os -T ldscript.ld
LDLIBS = -lm
OBJS = main.o sqrt.o prim.o fib.o
program: $(OBJS)
   $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@
```

3. (6) Napraviti **skriptu za povezivanje** te funkciju **`premjesti()`** tako da se program pripremi za učitavanje na adresu 0x10000, ali da ispravno radi tek kad se dijelovi premjeste na druge adrese, prema podjeli:

   a. sve iz datoteke `fib.c` u blok memorije na adresi 0x20000

   b. sve iz datoteke `prim.c` u blok memorije na adresi 0x30000

   c. sve iz datoteke `sqrt.c` u blok memorije na adresi 0x40000

   d. sve iz datoteke `main.c` u blok memorije na adresi 0x50000, osim funkcije `premjesti()` koja treba ostati tamo gdje je početno učitana (u bloku koji počinje s 0x10000).

   U skriptu ugraditi potrebne varijable koje koristiti u funkciji `premjesti()` koja se poziva prva, prije `main`, a koja treba premjestiti zadane dijelove na navedene adrese. Po potrebi dodatno označiti tu funkciju u kodu.

```
SECTIONS {
  fib_start = 0x10000;
  .fib 0x20000 : AT(fib_start)
  {
    fib.o(*)
  }
  fib_size = SIZEOF(.fib);

  prim_start = fib_start + fib_size;
  .prim 0x30000 : AT(prim_start)
  {
    prim.o(*)
  }
  prim_size = SIZEOF(.prim);

  sqrt_start = prim_start + prim_size;
  .sqrt 0x40000 : AT(sqrt_start)
  {
    sqrt.o(*)
  }
  sqrt_size = SIZEOF(.sqrt);
  premjesti_start = sqrt_start + sqrt_size;
  .premjesti premjesti_start: AT(premjesti_start)
  {
    main.o(.premjesti)
  }
  premjesti_size = SIZEOF(.premjesti);

  main_start = premjesti_start + premjesti_size;
  .premjesti 0x50000 : AT(main_start)
  {
    main.o(*)
  }
  main_size = SIZEOF(.main);
}
```

```
void premjesti()
__attribute__((section(".premjesti")))
{
  extern char
  fib_start, fib_size,
  prim_start, prim_size,
  sqrt_start, sqrt_size,
  main_start, main_size;
  char *od, *kamo;
  size_t koliko, i;

  koliko = (size_t) &fib_size;
  od = &fib_start;
  kamo = (char *) 0x20000;
  for (i = 0; i < koliko; i++)
    kamo[i] = od[i];

  koliko = (size_t) &prim_size;
  od = &prim_start;
  kamo = (char *) 0x30000;
  for (i = 0; i < koliko; i++)
    kamo[i] = od[i];

  koliko = (size_t) &sqrt_size;
  od = &sqrt_start;
  kamo = (char *) 0x40000;
  for (i = 0; i < koliko; i++)
    kamo[i] = od[i];

  koliko = (size_t) &main_size;
  od = &main_start;
  kamo = (char *) 0x50000;
  for (i = 0; i < koliko; i++)
    kamo[i] = od[i];
}
```

4. (3) Neki procesor ima integriran sklop za prihvat prekida. On se programira na način da se u 32-bitovni registar `IRQE` postavi broj čije jedinice predstavljaju omogućene prekidne ulaze (one koji se prihvaćaju) te da se u registar `IRQT` postavi adresa u memoriji gdje se nalazi tablica s adresama funkcija za obradu prekida. Ulazi su numerirani od 0 do 31. Npr. ako je bit 7 u registru `IRQE` postavljen u 1 onda će se prekid naprave koja je spojena na ulaz 7 prihvatiti te će se pozvati funkcija za obradu tog prekida koja se treba nalaziti na adresi `IRQT+7*sizeof(void*)`. Ostvariti **prekidni podsustav** sa sučeljima `void inicijaliziraj()` i `void registriraj_funkciju(int irq, void *funkcija)`. Dok se neka funkcija ne registrira za neki prekid on mora biti onemogućen u sklopu. Također, ako je argument `funkcija` jednak `NULL` u pozivu `registriraj_funkciju`, onda se zadani prekid treba zabraniti.

```
void *irqt[32]={0};

void inicijaliziraj()
{
   IRQE = 0;
   IRQT = irqt;
}
```

```
void registriraj_funkciju(int irq, void *funkcija) {
   irqt[irq] = funkcija
   if (funkcija != NULL)
         IRQE = IRQE | (1<<irq);
   else
         IRQE = IRQE & ~(1<<irq);
}
```

5. (2) Popraviti sljedeće makroe (ovdje ili na papirima):

```
#define MAX(X,Y)                  X > Y ? X : Y
//primjer poziva: t = MAX(i+1, j+k, k*t); //krivi primjer s 3 argumenta

#define POVECAJ(A,B,C,D)          A++ ; B++ ; C++ ; D = A + B + C ;
// primjer poziva: if (x > y)
//                     POVECAJ(x, y, *(z+j), w);
//                 else
//                     x = y;
```

```
#define MAX(X,Y)            ((X)>(Y)?(X):(Y)) //ili MAX(X,Y,Z) (rj. s 3 arg.)
#define POVECAJ(A,B,C,D)    do { (A)++;(B)++;(C)++;(D)=(A)+(B)+(C); } while(0)
```

6. (3) Programeri X i Y rade zajedno na nekom projektu za koji koriste alat git. U nekom trenutku sadržaj datoteke `radno` koja se nalazi u zajedničkom repozitoriju jest:

```
A
B
C
D
```

Programeri sada rade paralelno (lokalno, nad svojom kopijom repozitorija): X dodaje redak sa znakom `X` na početak datoteke, a Y mijenja redak sa znakom `C` u znak `Y`. Programer X tada prvi pokreće naredbe:

```
git add radno
git commit -m "+X"
git push
```

Programer Y nakon toga iste naredbe (uz `C=>Y` umjesto `+X` u komentaru naredbe `commit`).

a) Hoće li X uspjeti napraviti zadano bez grešaka? Ako ima grešaka što mora X napraviti da ih otkloni?

b) Hoće li Y uspjeti napraviti zadano bez grešaka? Ako ima grešaka što mora Y napraviti da ih otkloni?

c) Koji je konačni sadržaj datoteke `radno` (nakon otklanjanja svih grešaka i unosa obje promjene)?

a) X će uspjeti bez grešaka

b) Y neće uspjeti napraviti push; najprije treba napraviti pull koji će zahtijevati dodatni commit, jer se promjene spajaju s onima napravljenim od X-a, pa tek onda push

c) X A B Y D (svaki u svom redu)